BTC Embedded Systems
ISO 26262 compliant and highly automated Test Solutions for TargetLink

Dr. Udo Brockmeyer
Agenda

- About BTC
- Introduction
- Requirement-based Testing
- Back-to-back Testing
- Formal Specification and Formal Verification
- Conclusion

- Future Challenges for Automatic Test and Verification Tools
Agenda

- About BTC
- Introduction
- Requirement-based Testing
- Back-to-back Testing
- Formal Specification and Formal Verification
- Conclusion

- Future Challenges for Automatic Test and Verification Tools
BTC Embedded Systems AG

- **Company established in 1999**
- **BTC-ES Headquarter in Oldenburg (D)**
  - Subsidiaries in Munich and Berlin (D)
  - BTC Japan Co., Ltd.
  - Distributors in Sweden, India and South Korea

- **Mission Statement:**
  Our mission is to enable customers to increase product quality in a shortened design phase by introducing automatic test and verification technology to the model-based systems & software development process.

- **Main Customer Domains:**
  Automotive, Aerospace
Tool Vendor in Embedded Systems Domain:

Software Development: Automatic Testing & Formal Verification

- Domain: 95% Automotive
- Model Based: MathWorks Simulink & dSPACE TargetLink
- Strategic Partner: dSPACE
- BTC Tools: EmbeddedTester, EmbeddedValidator, EmbeddedSpecifier
- Standards: IEC 61508, ISO 26262

System Design: Automatic Testing

- Domain: Aerospace, Defense, Automotive and others
- Model Based: SysML with IBM Rhapsody
- Strategic Partner: IBM Rational
- BTC Tools: TestConductor, ATG
- IEC 61508, ISO 26262 and mainly DO 178B/C
Strategic Partnership with dSPACE

- BTC products are highly integrated into Simulink/TargetLink
- Common Marketing and Development Activities together with dSPACE
- Common Certified Tool-Chain to support IEC 61508 and ISO 26262 Standards
BTC Embedded Systems – Partial User List

- Toyota
- Honda
- Magneti Marelli
- MAN
- Nissan
- Denso
- Keihin
- MAN
- Magna
- Mitsubishi Motors
- Hitachi
- Jatco
- Ford
- Daimler
- Omron
- Suzuki
- Continental
- PSA Peugeot Citroën
- Fiat
- Porsche
- Claas
- Wabco
Agenda

- About BTC
- **Introduction**
- Requirement-based Testing
- Back-to-back Testing
- Formal Specification and Formal Verification
- Conclusion
1. Do you want to test your model or your code?

2. How much time do you spend with writing and executing test cases?

3. What if your PC could understand your requirements?
• Functional safety standard for automotive
• Provides an automotive safety lifecycle including software development and testing
• Explicitly addresses model based design and model based testing
BTC EmbeddedTester is certified as „Fit for purpose“ for ISO 26262

Reference Workflow for TargetLink + BTC EmbeddedTester available
Does my system behave as expected?

Did I test everything?

Is my selected scaling appropriate?

Do model and code always deliver the same results?

Is there dead code?

Are there test cases that violate a requirement?

Can a requirement ever be violated?

Requirement-based Testing

Back-to-Back Testing

Formal Verification
Model Based Development – 3 Test Methods

- **Requirement-based Testing** → **Back-to-Back Testing** → **Formal Verification**

**Development**

- **Textual Requirements** → **Functional Model** → **Implementation Model** → **C-Code**
  - Textual Spec e.g. in Doors → Simulink Model → TargetLink Model → ANSI C-Code

**Verification**

- **Test Cases** → **Simulation** → **Simulation** → **Simulation** → **Report** Incl. Coverage Information
  - e.g. SignalBuilder, Excel, .mat → MIL → MIL → SIL/PIL

**Purpose:** Show that model and code correctly implement requirements.
Model Based Development – 3 Test Methods

**Development**

- Functional Model
  - Simulink Model
- Implementation Model
  - TargetLink Model
- C-Code
  - ANSI C-Code

**Verification**

- Reference Simulation
  - MIL
- Back-to-Back Test
  - MIL
- Test Cases
  - Automatically generated
- Goal: High Structural Coverage

**Purpose**: Show that model and code are equivalent (structural testing)

- Requirement-based Testing
- Back-to-Back Testing
- Formal Verification
Model Based Development – 3 Test Methods

Development

Textual Requirements → Formalized Requirements
- Textual Spec e.g. in Doors
- Functional Model (e.g. Simulink Model)
- Implementation Model (e.g. TargetLink Model)
- C-Code (e.g. ANSI C-Code)

Verification

Formal Verification
- Simulation based formal verification
- Complete Analysis

Purpose: Show that requirements are not/cannot be violated.
• About BTC
• Introduction
• **Requirement-based Testing**
• Back-to-back Testing
• Formal Specification and Formal Verification
• Conclusion

• **Future Challenges for Automatic Test and Verification Tools**
**Problem:** 80% of development costs are spent identifying and fixing defects

**Solution:** Systematic and efficient unit test allows to discover errors earlier
Model-based Testing - Challenges

- Traceability between Tests and Requirements
- Code Coverage
- Handling of Calibration Parameters
- Reporting
- Debugging
- Model Coverage
- Test execution on Model and Production Code

**Problem:** A collection of not well integrated tools and scripts

**Solution:** Integrated Test Environment as „one-stop“ solution for model and code
Before creating or importing test cases, requirements can be imported in order to link and trace test cases to requirements.

Direct access to DOORS or PTC Integrity Databases
Create Test Cases → Execute Test Cases → Calculate Coverage

Create or import/export functional tests

Embedded Tester – Requirement-based Testing

BTC Embedded Tester: C:Work\Demo\PowerWindow\LET_Profiles\Empty_Profile.atgc

Analysis

BTC Embedded systems
Embedded Tester – Requirement-based Testing

Create Test Cases → Execute Test Cases → Calculate Coverage

Model Refinement

Model -> Automatic Code Generation

Simulink-MIL

TargetLink-MIL

Model

Import

Test Cases

Simulation

Comparing to reference

C-Code

Compilation

SIL

Obj-Code

PIL
1. Strong hierarchical approach

- Automatic analysis of model hierarchy
- Easily test and debug sub-functions in model and code
- No need to extract system under test manually

2. Grey Box

- Automatic detection of interface variables on all hierarchy levels
- Possibility to treat DISP variables as output
- Possibility to treat CAL variables as input
Embedded Tester – Test Execution

- Test Report Generation (HTML, PDF).
- Automatic comparison and setting of PASSED/FAILED.
- Direct link to test management for view or export for debugging.

### Test Report Generation

- Create Test Cases
- Execute Test Cases
- Calculate Coverage

### Testvector comparisons

#### Comparison #1
- **Name**: tv_ss3_tl_mil5
- **Length**: 1
- **Result**: PASSED
- **Reference Vector**: tv_ss3_tl_mil5 (length: 1)
- **Simulated Vector**: n.a.

#### Comparison #2
- **Name**: tv_ss3_tl_mil6
- **State**: 1
- **Reference Value**: -49654.0
- **Simulated Value**: 15882.0
- **Difference**: 65536.0

### Tolerance Definitions

<table>
<thead>
<tr>
<th>Check</th>
<th>Name</th>
<th>Kind</th>
<th>Data Type</th>
<th>LSB</th>
<th>Offset</th>
<th>Unit</th>
<th>Relative Tolerance</th>
<th>Absolute Tolerance</th>
<th>Relative Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>f2[1]</td>
<td>Outport</td>
<td>Int16</td>
<td>1</td>
<td>0</td>
<td>n.a.</td>
<td>0%</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Create Test Cases → Execute Test Cases → Calculate Coverage

Debugging Environment can be created for any hierarchy level!

Subsystem SimpleMinMaxTLM
Testvector Comparison Results

Tolerance Definitions

Testvector comparisons

Embedded Tester - Debugging

Embedded Tester: Back-to-back Report Viewer

Embedded Tester: Vector

Embedded Tester: Model

Embedded Tester: Simulink Debugger: model2verify

Debugging Environment can be created for any hierarchy level!
9.4.4 To evaluate the completeness of test cases and to demonstrate that there is no unintended functionality, the coverage of requirements at the software unit level shall be determined and the structural coverage shall be measured in accordance with the metrics listed in Table 14. If necessary, additional test cases shall be specified or a rationale shall be available.

### Table 14 — Structural coverage metrics at the software unit level

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1a Statement coverage</td>
<td>++</td>
</tr>
<tr>
<td>1b Branch coverage</td>
<td>+</td>
</tr>
<tr>
<td>1c MC/DC (Modified Condition/Decision Coverage)</td>
<td>+</td>
</tr>
</tbody>
</table>

NOTE 2 In the case of model-based development, software unit testing may be moved to the model level using analogous structural coverage metrics for models.
Reporting: Requirements Coverage Report

- Bi-directional traceability between requirements and test cases
- Identify untested requirements
- Identify violated requirements
### Reporting: Model Coverage Report

- Coverage information based on Simulink V&V Toolbox
- Coverage is cumulated for runs on different model hierarchy levels
- Intuitive Graphical Colouring of Simulink and Stateflow charts.

---

#### Model Coverage Report

**Global Coverage Statistics**

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Detailed Coverage Report</th>
<th>Graphical Model Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subsystem 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Test Cases**

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Total</th>
<th>Covered</th>
<th>Not Covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>100</td>
<td>73</td>
<td>27</td>
</tr>
<tr>
<td>Covered</td>
<td>73</td>
<td>73%</td>
<td></td>
</tr>
<tr>
<td>Not Covered</td>
<td>27</td>
<td>27%</td>
<td></td>
</tr>
</tbody>
</table>

**Coverage Information**

- Decision Coverage
  - Total: 12, Covered: 11, Not Covered: 1
  - Coverage: 92%
- Condition Coverage
  - Total: 12, Covered: 11, Not Covered: 1
  - Coverage: 92%
- MC/DC Coverage
  - Total: 16, Covered: 15, Not Covered: 1
  - Coverage: 94%

**Transition Decision Coverage**

<table>
<thead>
<tr>
<th>Transition Decision Coverage</th>
<th>Total</th>
<th>Covered</th>
<th>Not Covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>22</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>Covered</td>
<td>11</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td>Not Covered</td>
<td>11</td>
<td>50%</td>
<td></td>
</tr>
</tbody>
</table>

**Transition MC/DC Coverage**

<table>
<thead>
<tr>
<th>Transition MC/DC Coverage</th>
<th>Total</th>
<th>Covered</th>
<th>Not Covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Covered</td>
<td>0</td>
<td>n.a.</td>
<td></td>
</tr>
<tr>
<td>Not Covered</td>
<td>0</td>
<td>n.a.</td>
<td></td>
</tr>
</tbody>
</table>

**State Coverage**

<table>
<thead>
<tr>
<th>State Coverage</th>
<th>Total</th>
<th>Covered</th>
<th>Not Covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Covered</td>
<td>0</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Not Covered</td>
<td>0</td>
<td>0%</td>
<td></td>
</tr>
</tbody>
</table>
Global Code Coverage (Coverage Statistics, Condition, Decision, C/DC, MC/DC, Switch and Function Coverage)

Detailed Code Coverage (UID for test properties, links to the code and model parts)

Coloured Code Coverage (Source Code with Coloured Coverage Indication)

<table>
<thead>
<tr>
<th>Tests</th>
<th>Handled</th>
<th>Covered</th>
<th>Unreachable (n/inf)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement Coverage</td>
<td>16</td>
<td>16 88.9%</td>
<td>16 88.9%</td>
</tr>
<tr>
<td>Decision Coverage</td>
<td>4</td>
<td>3 75%</td>
<td>3 75%</td>
</tr>
<tr>
<td>Condition Coverage</td>
<td>0</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Condition/Decision Coverage</td>
<td>7</td>
<td>6 85.7%</td>
<td>6 85.7%</td>
</tr>
<tr>
<td>Modified Condition/Decision Coverage</td>
<td>7</td>
<td>6 85.7%</td>
<td>6 85.7%</td>
</tr>
</tbody>
</table>

```c
/* Requirements: Compute
Description: REQ_2
Document: Simple
Location: B1
MinMax: Module/F1/mi

2 T ? 213 if ((Sa2_max) < (Sa1_In) 213b
2
214 /* TL_AUTO_COMMENT B
215 Sa2_min = Sa2_max;
216 }
217 else {
218 /* TL_AUTO_COMMENT B
219 Sa2_min = Sa1_In3;
220 }
221 }
```
Agenda

- About BTC
- Introduction
- Requirement-based Testing
- Back-to-back Testing
- Formal Specification and Formal Verification
- Conclusion

- Future Challenges for Automatic Test and Verification Tools
9.4.4 To evaluate the completeness of test cases and to demonstrate that there is no unintended functionality, the coverage of requirements at the software unit level shall be determined and the structural coverage shall be measured in accordance with the metrics listed in Table 14. If necessary, additional test cases shall be specified or a rationale shall be available.

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL A</th>
<th>ASIL B</th>
<th>ASIL C</th>
<th>ASIL D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a Statement coverage</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>1b Branch coverage</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1c MC/DC (Modified Condition/Decision Coverage)</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>++</td>
</tr>
</tbody>
</table>

NOTE 4 For model-based development, software unit testing can be carried out at the model level followed by back-to-back comparison tests between the model and the object code. The back-to-back comparison tests are used to ensure that the behaviour of the models with regard to the test objectives is equivalent to the automatically-generated code.
EmbeddedTester – Back-to-Back Testing

Model

Simulink-MIL

Model Refinement

Model

TargetLink-MIL

Automatic Code Generation

C-Code

Compilation

SIL

Obj-Code

PIL

Stimuli Vectors

~100% Coverage

Generate automatically

Simulation

Stimuli Vector Generation

Recording Reference Outputs

Back to Back Testing

Back to Back Testing
BTC EmbeddedTester - Test Goals

- **Structural Coverage Goals**
  - Statement Coverage
  - Condition Coverage
  - Decision Coverage
  - Switch-Case-Coverage
  - Function-Call-Coverage
  - Modified Condition/Decision Coverage
  - Domain Coverage

- **Robustness Analyse**
  - Relational Operators
  - Division-by-Zero
  - Down-casting
  - Range Violation
  - Unreachable

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Relational Operator (GEQ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique ID</td>
<td>TC92</td>
</tr>
<tr>
<td>File</td>
<td>CL_Controller.c</td>
</tr>
<tr>
<td>Line</td>
<td>1032</td>
</tr>
<tr>
<td>Shared With Functions</td>
<td>CentralLockingTL_OK/CL_Controller/Subsystem/CL_Controller</td>
</tr>
<tr>
<td>TargetLink Blocks</td>
<td>(CL_Controller/Aggregate_Behavior/Motor_DriverDoor)</td>
</tr>
<tr>
<td>Expression</td>
<td>Ch1_Motor_DriverDoor ctr&gt;=3</td>
</tr>
</tbody>
</table>

Example

Properties
- RO:92:1 operation became true
- RO:92:2 operation became false
- RO:92:3 (left-right) == -1
- RO:92:4 (left-right) == 0
- RO:92:5 (left-right) == 1
Embedded Tester – Use cases for Back-to-Back Testing

1. **TargetLink Model vs. Code**
   
   - Typical use case in order to compare Simulink Model (MIL) to C-Code (SIL/PIL)

2. **Simulink Model vs. TargetLink Model**
   
   - Useful in case an original Simulink Model (e.g. provided by a customer) is modified to become a TargetLink Model (e.g. because of unsupported blocks or for optimization reasons)

3. **Current Model Version vs. Previous Model Version**
   
   - Automatic regression test between model versions

4. **Current Matlab/TargetLink Version vs. Previous Version**
   
   - The EmbeddedTester Migration Suite allows to verify automatically, that the migration to a new Matlab and/or TargetLink version does not change the behavior of models and production code.
Embedded Tester – Test Automation

Summary Report
powerwindow_tl_v01

General Information

<table>
<thead>
<tr>
<th>Creator</th>
<th>markusg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Date Time</td>
<td>19-Jan-2014 16:25:36</td>
</tr>
<tr>
<td>End Date Time</td>
<td>19-Jan-2014 16:31:23</td>
</tr>
<tr>
<td>Report Location</td>
<td>C:\Work\Demos\PowerWindow_2014\etta\report\etta_20140110_162536\ETTA_SummaryReport.html</td>
</tr>
</tbody>
</table>

Test Automation Steps

<table>
<thead>
<tr>
<th>Step</th>
<th>Status</th>
<th>Result Details</th>
<th>SavedProfile</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Profile Creation</td>
<td>Completed with Warnings</td>
<td>Profile Report</td>
<td>etta_empty.atgcv</td>
<td>00:00:37</td>
</tr>
<tr>
<td>2. SV Generation</td>
<td>Completed</td>
<td>Analysis Report</td>
<td>etta_sv_all.atgcv</td>
<td>00:03:28</td>
</tr>
<tr>
<td>3. Reference Simulation TL-MIL</td>
<td>Completed with Warnings</td>
<td></td>
<td>etta_Ref_TL_MIL.atgcv</td>
<td>00:01:19</td>
</tr>
<tr>
<td>4. Back-to-back Test TL-MIL vs SIL</td>
<td>Completed</td>
<td>Back-to-back Report SIL</td>
<td>etta_TI_MIL_vs_SIL.atgcv</td>
<td>00:00:10</td>
</tr>
</tbody>
</table>

TestAutomation finished.
Agenda

• About BTC
• Introduction
• Requirement-based Testing
• Back-to-back Testing
• **Formal Specification and Formal Verification**
• Conclusion

• Future Challenges for Automatic Test and Verification Tools
A Hierarchy of Notation Methods is defined

The more safety critical a function is, the more formal the notation and verification is recommended
Problem:
- A testcase only represents one possible path through the system
- It is impossible to cover all paths with test cases

Solution:
- Model checking analyses all possible paths and guarantees a bug-free system
Challenges when specifying requirements in a formal way

- **Problem 1**: Some languages that might be used to express requirements are not formal

- **Problem 2**: Formal methods are often considered to be too mathematical and too difficult to learn

- **Example of a formal specification in LTL**

  
  \[
  S \rightarrow x. (\diamond [x+n,x+n] \text{TRUE}) \rightarrow \square (y. (\diamond TSE \rightarrow (T C U_{y+\text{min},y+\text{max}} \text{ TEE})) \rightarrow z. (\diamond [z+\text{min},z+\text{max}] \text{ ASE} \land (\diamond [z+\text{min},z+\text{max}] \text{ ASE} \rightarrow r. (A C U_{r+\text{min},r+\text{max}} \text{ AEE}))))
  \]

- **Solution**: A tool that allows engineers to take their textual requirements and intuitively derive semi-formal and formal notations
Intuitive formalization process thanks to Pattern library in EmbeddedSpecifier

Non-ambiguous representation helps to improve the quality of requirements

Formalized requirements are later used in Formal Verification process

“Proven in use” in Automotive and Avionics Industry
Simulation-based Formal Verification

Solution 1: Co-Simulation

BTC EmbeddedSpecifier

Export

Test Environment

Requirement Observer

Test Cases

System Under Test

Requirement Status

Fullfilled / Violated

Solution 2: Offline-Verification

BTC EmbeddedValidator BASE

Import

Test Data

Formal Specification

P R O O F

Test case XY violated
Requirement 5

Requirement fullfilled
EmbeddedValidator - Method

- Formal Specification
- Formal Verification
  - Simulation-based
  - Complete Analysis

- dSPACE TargetLink
- BTC EmbeddedSpecifier
- Safety Requirements
- BTC EmbeddedValidator
- TargetLink Code
- Formal Requirement

Code does not fulfill the requirement + Counter Example

- Code fulfills requirement
Agenda

• About BTC
• Introduction
• Requirement-based Testing
• Structural Testing (Back-to-back Testing)
• Formal Specification and Formal
• Conclusion

• Verification
Future Challenges for Automatic Test and Verification Tools
Conclusion – 3 Test Methods

- **Requirements-based testing** usually finds about 20-40% of the problems.

- additional 30-40% of the software problems can be directly found by using **structural testing** and **back-to-back** comparison (Small effort due to automatic test case generation and test execution)

- **Formal verification** is especially relevant for testing of safety-critical software.

- Combination of test methods is recommended to achieve high quality

- ISO 26262 provided guidelines on the test methods to be used for each ASIL

* Metrics Source: German Automotive OEM – Model Based Project.
### Requirement-based Testing
- Create and manage functional test cases
- Import requirements (e.g. DOORS) and link test cases to requirements
- Automatic MIL/SIL/PIL test execution on any hierarchy level
- Automatic test evaluation
- Coverage Reports for Requirements, Model and Code
- Generation of a debug environment in Simulink or MSVC

### Back-to-Back Testing
- Automatic test case generation for full structural coverage on production code
- Execution of Back-to-Back Test
- User-defined structural test goals (Equivalence Class, Boundary, etc.)
- Includes all Features from BTC EmbeddedTester BASE

### Formal Verification
- Formal Specification based on pattern library
- Import of informal requirements from DOORS, PTC Integrity, Excel, etc.
- Intuitive creation of semi-formal and formal requirements
- Full traceability of requirements

#### BTC EmbeddedSpecifier
- Simulation-based Formal Verification
- Formal Verification using ModelChecking Technology

#### BTC EmbeddedTester BASE

#### BTC EmbeddedTester

#### BTC EmbeddedValidator BASE

#### BTC EmbeddedValidator
Conclusion

- Requirement-Based-Testing for Simulink/TargetLink-Models and C-Code
- Highly integrated with dSPACE TargetLink
- Connection to Requirements-Management Tools like DOORS
- Automatic test execution (MIL/SIL/PIL) on any hierarchy level
- Automatic generation of debugging environments
- Integrated coverage measurement (requirements coverage, model coverage, code coverage)
Efficiency Improvements

Customer experiences show a decreasing overall test effort by 50 to 70%

- Test case generation effort for maximal structural coverage could be minimized by 90%!
- Effort savings of up to 70% during test execution and test evaluation phases
- Half of the debugging effort could be saved by using interactive and automatic debugging tool support

Quality Improvements

Improved „Maturity Gates“

- In average the MC/DC Coverage rates are 30% higher in contrast to manual test approaches

Process Improvement

ISO26262 Certified Tool-Chain with BTC EmbeddedTester supports Standard conform development and test process
"MAN Nutzfahrzeuge AG successfully uses EmbeddedTester as a standard Automatic Test Environment for the leading AutoCode Generator TargetLink in the Model Driven Development of series-production Power Train applications. The automatic test generation, execution, analysis and debug capabilities of EmbeddedTester is one important key to fulfill the high efficiency and quality levels of MAN Nutzfahrzeuge AG, under the permanent time-to-market pressure."

Stefan Teuchert,
Head of the Department Software-Development and Base Technologies, MAN Nutzfahrzeuge AG (Munich)
Can be intuitively used by requirements engineers or function developers to create semi-formal and formal requirements

(Semi-)Formal and unambiguous representation helps to improve requirements quality

Import from (and traceability to) RM-tools like DOORS and PTC Integrity

Automatic translation into machine readable specification in order to use requirements in the verification process

Directly supports safety standards like IEC 61508 & ISO 26262

Highly recommended for Safety Critical Applications
Simulation Based Formal Verification allow crosschecking of all test cases against all requirements.

Integrated reporting with full traceability to original requirements.

Flexible concept for supporting different test environments:
- Support for Simulink models and TargetLink models available.
- Support for dSPACE VEOS and dSPACE HIL-Systems planned for 2014.
- Other test architectures can be easily integrated.
✓ Mathematical proof that Code can never violate a requirement

✓ Check requirements on any hierarchy level of a model

✓ For requirements that can be violated, a counter example and a debugging model is generated automatically
Agenda

• About BTC
• Introduction
• Requirement-based Testing
• Structural Testing (Back-to-back Testing)
• Formal Specification and Formal
• Conclusion

• Future Challenges for Automatic Test and Verification Tools
Future Challenges for Automatic Test and Verification Tools

1. Optimized “off-the-shelf” integration into customer processes and tool chains

   Solution:
   • Open interfaces to connect existing customer data-backbone of MBD Process
   • any V&V activity in the MBD Process is related to existing data-backbones providing full traceability

2. Dedicated customer workflow support and improved usability

   Solution:
   • Generalized test artifact management via V&V-Tools
   • Focused use case perspectives incl. wizard functionality
   • Unified graphical user interface for all use V&V cases

3. Providing a framework for a complete tool-chain qualification for safety-critical applications

   Solution:
   • One general validation suite and qualification approach

4. Dealing with different complexities

   Solution:
   • Improved test and verification engines - runtime and coverage
   • Sophisticated analysis technologies - Floating-Point Applications
   • Extended REQ Specification Method – Usability & Expressiveness

One common technology platform is necessary to provide an efficient & effective customer solution
BTC EmbeddedPlatform Overview

BTC EmbeddedPlatform
Common Architecture/Interface Data-base and BTC-Core Functionality

BTC EmbeddedTester
BTC EmbeddedSpecifier
BTC EmbeddedValidator
Engines

• VIS (Colorado; BDDs)
• MiniSat (open source, SAT)
• cbmc (Oxford Uni and Carnegie Mellon; SAT)
• “BTC-engine” (Oldenburg; several heuristics)

Open Challenges

• Ability to cope
  • with control models and control programs containing non-linear equations
  • with growing demands regarding floating point precision
• Continuous growth of model and program complexity
Thank you.